plain concepts

PowerShell Core

- Operadores, variables, cadenas, arrays…

OPERADORES, VARIABLES, CADENAS, ARRAYS…

Variables

- Usamos el character $ para crear y usar variables `$myVariable`
- Puede tener letras, números, espacios y guiones bajos `${My Variable}`
- No persiste una vez salimos de la Shell
  - New-Variable
  - Set-Variable
  - Get-Variable
  - Clear-Variable
  - Remove-Variable
- Podemos forzar el tipo – [int]$var

plain concepts

Nota: El símbolo $ no es parte del nombre de la variable, es una marca para acceder al contenido de la variable

Type	Description
[int]	32-bit signed integer
[long]	64-bit signed integer
[string]	Fixed-length string of Unicode characters
[char]	A Unicode 16-bit character
[byte]	An 8-bit unsigned character
[bool]	Boolean True/False value
[decimal]	An 128-bit decimal value
[single]	Single-precision 32-bit floating point number
[double]	Double-precision 64-bit floating point number
[datetime]	Date and Time
[array]	An array of values
[hashtable]	Hashtable object

OPERADORES, VARIABLES, CADENAS, ARRAYS…

Arrays

- Array vacío:
  - $miArray = @()
- Sintaxis Explícita:
  - $miArray = @(1,"Hola",3.5,“Mundo")
- Sintaxis Implícita
  - $miArray = 1,"Hola",3.5,“Mundo"
- Operador de rango:
  - $miArray = (1..7)
- Añadir Elementos al Array:
  - $miArray += “NuevoElemento”

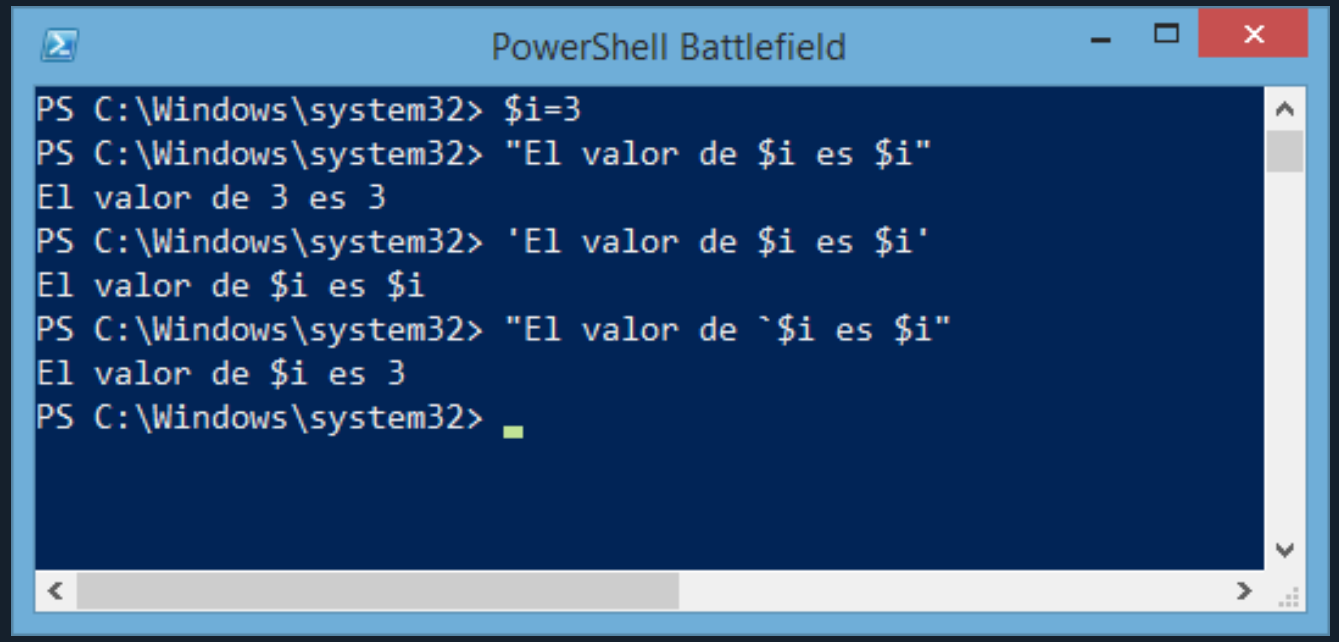
```
PS C:\Windows\system32> $array=1,2,3,4
PS C:\Windows\system32> $array[3]
4
PS C:\Windows\system32>
PS C:\Windows\system32> $array=(1..7)
PS C:\Windows\system32> $array[6]
7
PS C:\Windows\system32>
```

plain concepts

OPERADORES, VARIABLES, CADENAS, ARRAYS…

Cadenas: Uso de las “Comillas”

- Con las dobles comillas se resuelven todas las variables
- Con las comillas simples no hay sustitución
- Get-Help About_Quoting_Rules
- Con el acento grave forzamos la no resolución de variables individuales

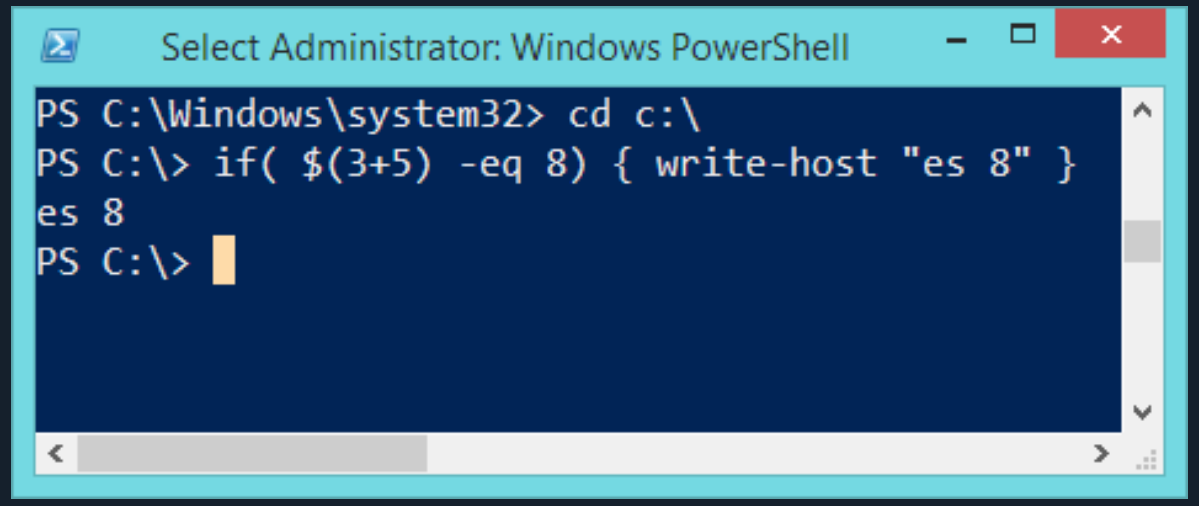


plain concepts

OPERADORES, VARIABLES, CADENAS, ARRAYS…

Sub-Expresiones

- Devuelve propiedades específicas de un objeto
- Puede constar de multiples secciones
- Cada sección contribuye al valor devuelto
- Podemos evaluar expresiones en una línea sin necesitar variables intermedias
- Útil para extraer valores de objetos o aplicarles un procesamiento previo a su evaluación

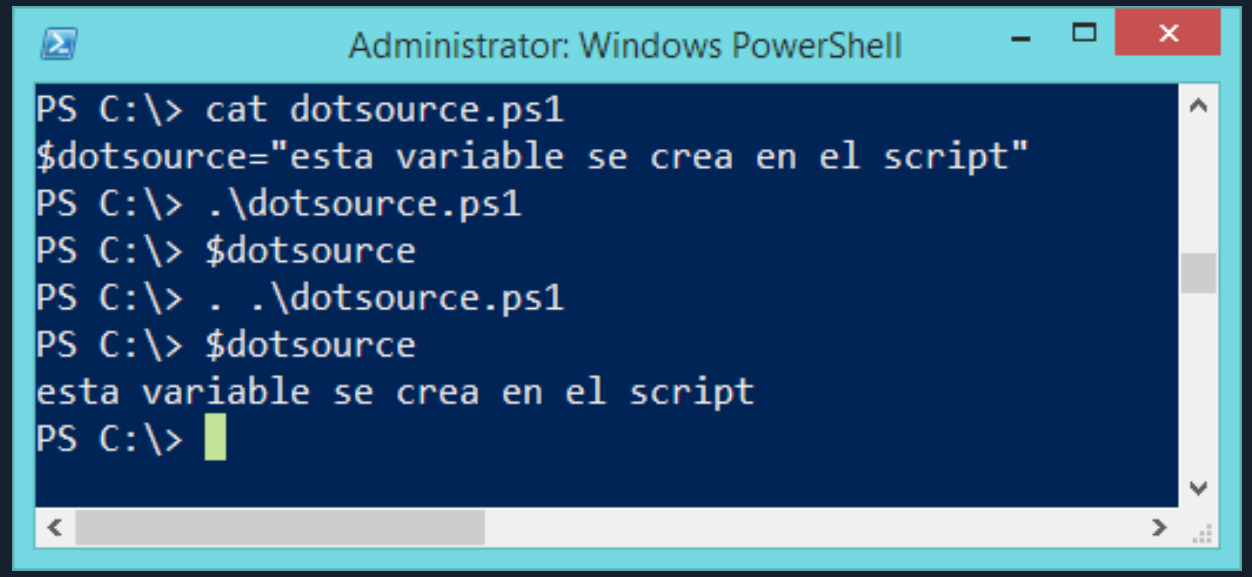


plain concepts

OPERADORES, VARIABLES, CADENAS, ARRAYS…

DotSourcing

- El operador "." ejecuta un script en el ámbito de ejecución activo.
- Por defecto, cuando se ejecuta un script, éste lo hace en un nuevo ámbito de ejecución.
  - Cuando el script finalize, las variables creadas desaparecen.
- Al ejecutar scripts en el ámbito actual, las variables creadas permanecen en el ámbito actual



plain concepts

plain concepts

Proveedores de PowerShell

- Otra forma de interactuar

PROVEEDORES

- Permiten que cualquier tipo de dato sea expuesto a powershell como un Sistema de archivos, como si se hubiese montado una unidad
- Son programas basados en .NET
- Cada proveedor proporciona cmdlets para manejar los datos

```
Provider      Drive         Data store
--------      -----         ----------
Alias         Alias:        Windows PowerShell aliases

Certificate   Cert:         x509 certificates for digital signatures

Environment   Env:          Windows environment variables

FileSystem    *             File system drives, directories, and files

Function      Function:     Windows PowerShell functions

Registry      HKLM:, HKCU:  Windows registry

Variable      Variable:     Windows PowerShell variables

WSMan         WSMan:        WS-Management configuration information

* The FileSystem drives vary on each system.
```

plain concepts

PROVEEDORES: CERTIFICADOS

Administrator: Windows PowerShell

```
PS HKLM:\SOFTWARE> cd cert:
PS Cert:\> ls


Location   : CurrentUser
StoreNames : {TrustedPublisher, ClientAuthIssuer, Root, UserDS...}

Location   : LocalMachine
StoreNames : {TrustedPublisher, ClientAuthIssuer, Remote Desktop, Root...}



PS Cert:\> cd .\\LocalMachine\My
PS Cert:\LocalMachine\My> ls


    Directory: Microsoft.PowerShell.Security\Certificate::LocalMachine\My


Thumbprint                                Subject
----------                                -------
E4FE22296833C54A8CAEF98445DFA7448F12722C  CN=localhost
CC92344A44139379B7B9831100BE49973483167D  CN=AMARQUEZ-HP.plainconcepts.com
```

plain concepts

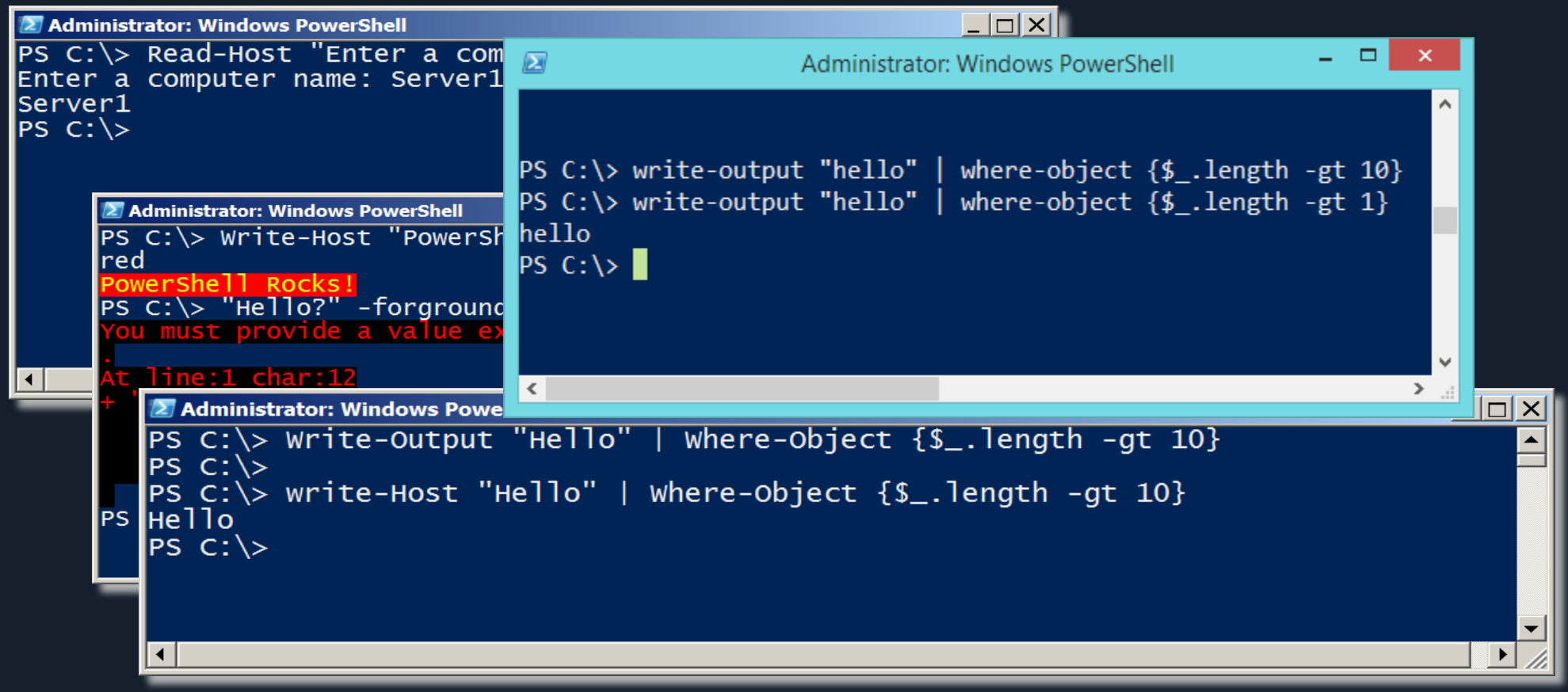
PowerShell Scripting

- Scripting básico

SCRIPTING

Back to basics: Entrada/Salida Interactiva

- Read-Host
- Write-Host
- Write-Output

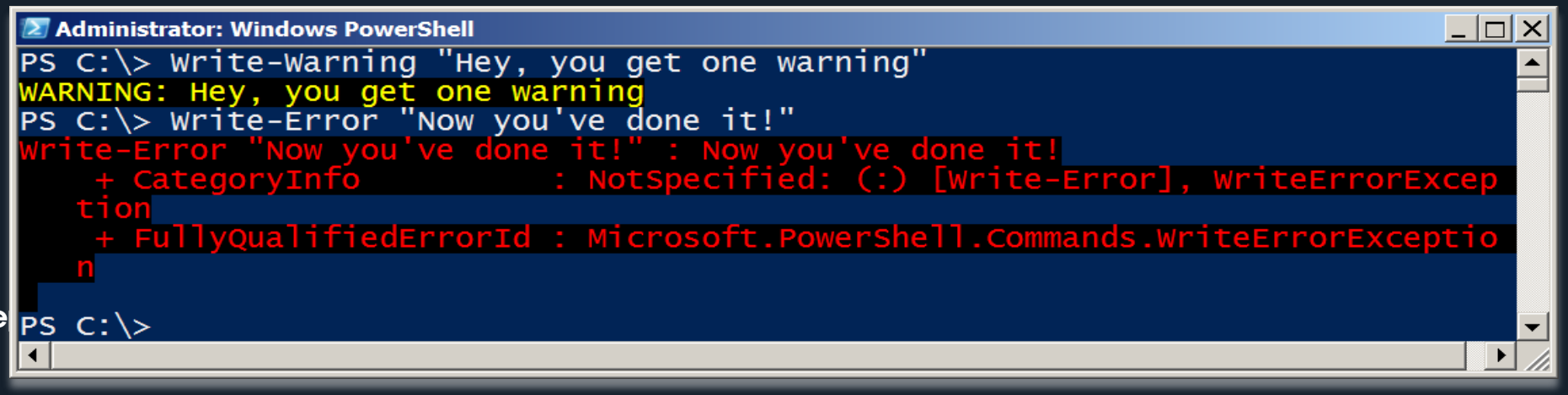


plain concepts

SCRIPTING

Otras salidas interesantes para scripts y automatización

- Write-Warning
- Write-Verbose
- Write-Debug
- Write-Error



plain conce

SCRIPTING

Objetivos de seguridad de PowerShell

- Por defecto está bloqueada la ejecución de Scripts
- Los archivos .Ps1 se pueden editar con el Bloc de Notas
- Es necesario escribir la ruta para ejecutar un script (./nombreScript.Ps1)

SCRIPTING

Política de ejecución

- Por defecto, PowerShell no permite la ejecución de ningún script.
- Get/Set-ExecutionPolicy
  - Restricted
  - Unrestricted
  - RemoteSigned
  - …
- Se puede establecer por GPO

plain concepts

SCRIPTING

Ejecutando Scripts

Administrator: Windows PowerShell

```
PS C:\> C:\Scripts\test.ps1
Hello world!
PS C:\> cd scripts
PS C:\scripts> .\test.ps1
Hello world!
PS C:\scripts> . .\test.ps1
Hello world!
PS C:\scripts> Get-Fun
MVA and Scripting with PowerShell Rocks!
PS C:\scripts>
```

plain concepts

DotSourcing!

SCRIPTING

Parametrizando nuestros comandos

```
1  #Choose values to replace with variables
2
3  $ComputerName='Client'
4  $Drive='c:'
5
6  Get-WmiObject -class Win32_logicalDisk -Filter "DeviceID='$Drive'" `
7      -ComputerName $ComputerName
```

- Añade flexibilidad a las acciones ejecutadas.
- Incrementa reusabilidad

plain concepts

SCRIPTING

Object Members y Variables

- Con Get-Member (gm) sacamos el TypeName, Methods y Properties de un Objeto.
- Podemos usar las variables para trabajar con los Objetos.

```
#Object Members and variables
#Variables are very flexible
$Service=Get-Service -Name bits
$Service | GM
$Service.Status
$service.Stop()
$Msg="Service Name is $($service.name.ToUpper())"
$msg
#Working with multiple objects
$Services=Get-Service
$services[0]
$services[0].Status
$Services[-1].Name
"Service Name is $($services[4].DisplayName)"
"Service Name is $($services[4].name.ToUpper())"
```

plain concepts

SCRIPTING

Bifurcaciones

```
If ($this -eq $that) {
    # commands
} elseif ($those -ne $them) {
    # commands
} elseif ($we -gt $they) {
    # commands
} else {
    # commands
}
```

```
# Switch can be easier to maintain than If statement
#and can provide additional features

Switch ($status) {
    0 { $status_text = 'ok' }
    1 { $status_text = 'error' }
    2 { $status_text = 'jammed' }
    3 { $status_text = 'overheated' }
    4 { $status_text = 'empty' }
    default { $status_text = 'unknown' }
}
```

SCRIPTING

Bucles

```
# Do loop
$i= 1
Do {
    Write-Output "PowerShell is Great! $i"
    $i=$i+1 # $i++
} While ($i -le 5) #Also Do-Until

# While Loop
$i=5
While ($i -ge 1) {
    Write-Output "Scripting is great! $i"
    $i--
}
```

```
# Foreach - used often in our scripting for today
$services = Get-Service
ForEach ($service in $services) {
  $service.Displayname
}

#For loop
For ($i=0;$i -lt 5;$i++) {
 #do something
}

#Another way
1..5 | ForEach-Object -process {
     Start calc
}
```

plain concepts

SCRIPTING

La ISE (Integrated Scripting Environment)

Administrator: Windows PowerShell ISE

File Edit View Tools Debug Add-ons Help

```
PS C:\> Get-WmiObject -Class win32_LogicalDisk -Filter "DeviceID='C:'" -ComputerName localhost
```

Administrator: Windows PowerShell ISE

File Edit View Tools Debug Add-ons Help

Script.ps1

```
1
2   Get-WmiObject -Class win32_LogicalDisk -Filter "DeviceID='C:'" -ComputerName localhost |
3       Select-Object -Property DeviceID,
4       @{n='SizeGB';e={$_.Size / 1GB -as [int]}},
5       @{n='FreeGB';e={$_.Freespace / 1GB -as [int]}} |
6
```

plain concepts

SCRIPTING

Añadiendo parámetros a nuestro Script

Administrator: Windows PowerShell ISE

File Edit View Tools Debug Add-ons Help

Script.ps1 | Script1a.ps1 | Script2.ps1

```
1  Param(
2      $ComputerName='localhost'
3  )
4
5  Get-WmiObject -Class win32_LogicalDisk -Filter "DeviceID='C:'" -ComputerName $ComputerName |
6      Select-Object -Property DeviceID,
7      @{n='SizeGB';e={$_.Size / 1GB -as [int]}},
8      @{n='FreeGB';e={$_.Freespace / 1GB -as [int]}}
```

```
PS C:\> .\Script1a.ps1 -
```

ComputerName | [Object] ComputerName

SCRIPTING

Documentando el Script

- Comentarios de código
- Se comporta como un bloque único de comentario
- Tiene varias secciones
  - .Synopsis
  - .Description
  - .Parameter
  - …
- Se integra totalmente con Get-Help

Administrator: Windows PowerShell ISE

File Edit View Tools Debug Add-ons Help

Script.ps1 | Script1b.ps1 | Script2.ps1 | Script1a.ps1

```
1  <#
2  .Synopsis
3  This gets the current disk size and freespace.
4  .Description
5  This is the long description
6  .Parameter ComputerName
7  This is the computer (local or remote) you want the diskspace from.
8  .Example
9  PS C:\> .\Script.ps1 -ComputerName <Computer>
10 Getting the diskspace of a remote computer
11 #>
12 Param(
13     $ComputerName='localhost'
14 )
15
16 Get-WmiObject -Class win32_LogicalDisk -Filter "DeviceID='C:'" -ComputerNar
17     Select-Object -Property DeviceID,
18     @{n='SizeGB';e={$_.Size / 1GB -as [int]}},
19     @{n='FreeGB';e={$_.Freespace / 1GB -as [int]}}
```

plain concepts

SCRIPTING

Transforma tu script en una función

- Modulariza la funcionalidad.
- [CmdletBinding()] indica que la función se comporte como un cmdlet

Administrator: Windows PowerShell ISE

File Edit View Tools Debug Add-ons Help

Script.ps1 | Script1b.ps1 | Script3.ps1* | Script1a.ps1

```
Function Get-DiskInfo{

    [CmdletBinding()]

     Param(
         [Parameter(Mandatory=$true)]
         $ComputerName='localhost'
     )

     $Disk=Get-WmiObject -Class win32_LogicalDisk -Filter "DeviceID='C:'" -ComputerName $Compute

     $Property=@{
         'Computer'= $ComputerName;
         'TotalSpace'= $Disk.size / 1gb -as [int];
         'FreeSpace' = $Disk.freespace / 1gb -as [int]}

     $Obj=New-Object -TypeName PSObject -Property $Property
     Write-Output $Obj
}
```

plain concepts

SCRIPTING

Documentando tu función

```
function Get-SecureString (){
<#
.SYNOPSIS
Lee una contrase►a desde teclado y la devuelve como un SecureString

.DESCRIPTION
Lee una contrase►a desde teclado y la devuelve como un SecureString

.NOTES
#>
    return read-host -Prompt "Password" -AsSecureString | ConvertFrom-SecureString
}
```

plain concepts

plain concepts

Gestión de Errores

GESTIÓN DE ERRORES

Control de errores

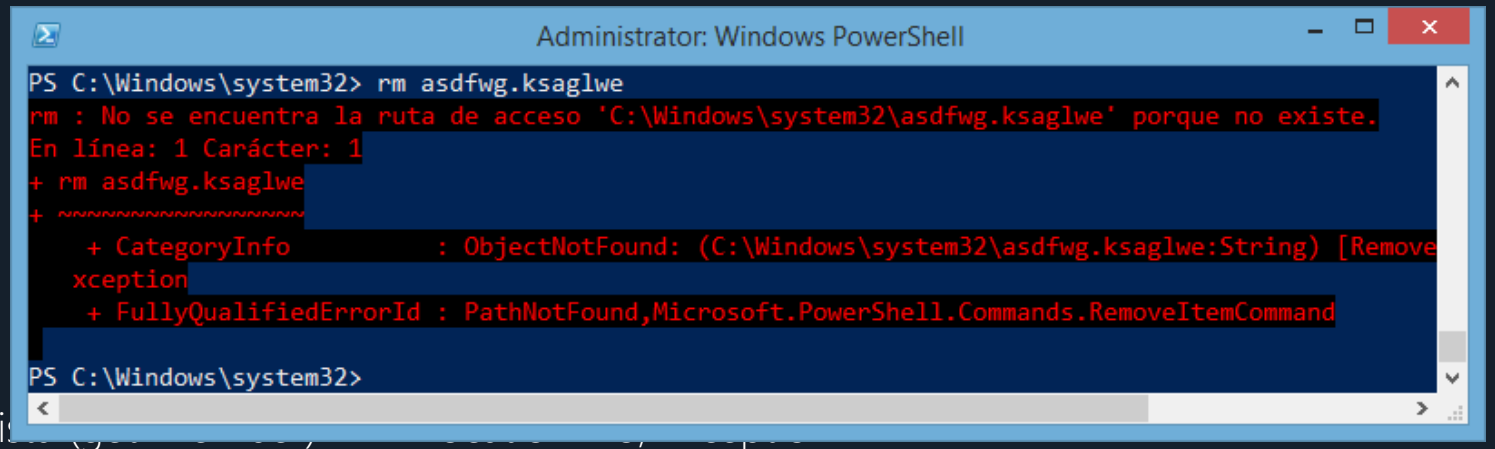
- Cuando un commando de PowerShell encuentra un error que no termina la ejecución, comprueba el valor de la variable por defecto **$ErrorActionPreference**:
  - Continue – Es el valor por defecto. Muestra el error y sigue ejecutándose.
  - SilentlyContinue – Sigue ejecutándose pero no muestra el error.
  - Stop – Muestra el error y para la ejecución.
  - Inquire – Muestra un mensaje para que el usuario elija que acción tomar.

GESTIÓN DE ERRORES

Control de errores

- Dos tipos:
  - Errores terminales: Finalizan la ejecución del script. Ej: Nulls
  - Errores no-terminales: Dan un aviso de error, pero el script sigue funcionando. Ej: File-not-found

- Variable $error
  - Almacena los errores encontrados.
  - Es Global
  - Tiene más de lo que se ve a simple vi



plain concepts

GESTIÓN DE ERRORES

Bloque try..catch

- Misión: Capturar los errores terminales en el momento de producirse
- Añade robustez al código
- Permite recuperarse del error
- Finally es un bloque opcional que se ejecuta *siempre*, independientemente de si hubo error o no
- Para capturar errores no terminales, marcar $erroractionpreference como **Stop**
- **Se pueden capturar excepciones específicas**

```
1   $Computer='notonline'
2  Try{
3       $os=Get-Wmiobject -ComputerName $Computer -Class Win32_OperatingSystem `
4        -ErrorAction Stop -ErrorVariable CurrentError
5
6  }
7  Catch{
8       Write-warning "You done made a boo-boo with computer $Computer"
9  }
```

plain concepts